NAS2-11530

# The Science of Computing -- 1985

*Peter J. Denning*

September 1985

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 85.12

(NASA-CR-187281) THE SCIENCE OF COMPUTING: 1985 (Research Inst. for Advanced Computer Science) 44 p

N90-71372

Unclas
00/61 0295397

RIACS

Research Institute for Advanced Computer Science

# The Science of Computing -- 1985

*Peter J. Denning*

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 85.12
September 1985

Because computer science affects every other scientific discipline, the editors of *American Scientist* felt it appropriate that a journal that spans the disciplines should have a regular column examining this field and its pervasive influence. They invited me to write such a column. I undertook this task beginning with the issue of January 1985. This report collects the texts of the six columns for 1985 into a single place.

The column focuses on fundamental issues in computer science that are likely to be of interest to scientists in other disciplines. Article No. 1 introduced computer science by pointing out the principal areas of the discipline and their fundamental questions. Articles Nos. 2 and 3 dealt with computer networks, a subject of increasing interest in the scientific community this year because of the NSFNET project. Articles Nos. 4 and 5 dealt with parallel computation, a subject of great interest among scientists who rely on massive computations. Article No. 6 discussed the arbiter problem, a basic limitation on the ability of physical circuits to select between two near-simultaneous events within a bounded time interval; this problem interested me because it is fundamental, physical, and not deeply mathematical.

I am indebted to *American Scientist* editor Michelle Press for her enthusiastic support and encouragement.

Work reported herein was supported in part by Contract NAS2-11530 from the National Aeronautics and Space Administration (NASA) to the Universities Space Research Association (USRA).

# What is Computer Science?

*American Scientist 73*
No. 1 (Jan-Feb 1985)

*The discipline of Computer Science has deep roots in mathematics, engineering, and logic. The core of the field can be subdivided into eleven areas, which address fundamental questions and have major accomplishments. Because the fundamental question underlying all of computer science -- "What can be automated?" -- will not soon be answered, the discipline will endure.*

Computer Science is the body of knowledge dealing with the design, analysis, implementation, efficiency, and application of processes that transform information. The fundamental question underlying all of computer science is, "What can be automated?" *(1)* This discipline was born in the mid-1940s with the invention of the stored-program electronic computer and has grown rapidly ever since.

Computer science has deep roots in mathematics, engineering, and logic. For several thousand years, a principal concern of mathematics has been calculation. Many models of physical phenomena have been used to derive equations whose solutions yield predictions of those phenomena -- for example, calculations of orbital trajectories, weather forecasts, and fluid flows. Many general methods for solving such equations have been devised -- for example, algorithms for systems of linear equations, differential equations, and integrating functions. For almost the same period, a principal concern of engineering has been calculations that aid in the design of mechanical systems. Examples include algorithms for evaluating stresses in static objects, calculating momenta of moving objects, and measuring distances much larger or smaller than our immediate perception.

One product of the long interaction between engineering and mathematics has been mechanical aids for calculating. Some surveyors' and navigators' instruments date back a thousand years. Pascal and Leibniz built arithmetic calculators in the middle 1600s. In the 1830s, Babbage conceived of an "analytical engine" that could mechanically and without error evaluate logarithms, trigonometric functions, and other general arithmetic functions. His machine, never completed, served as an inspiration for later work. In the 1920s, Bush constructed an electronic analog computer for solving general systems of differential equations. By the 1920s electromechanical calculating machines capable of addition, subtraction, multiplication, division, and square root were available. The electronic flip-flop provided a natural bridge from these machines to digital versions with no moving parts.

Logic is also an old discipline, concerned with criteria of validity of inference and formal principles of reasoning. Since the days of Euclid, it has been a tool for rigorous mathematical and scientific argument. By 1830, it was obvious that all the known deductive systems were incomplete because paradoxes could always be found. This led to a century-long search for a "complete" deductive system – within which it would be possible to determine mechanically whether or not any give statement is either true or false. In 1931, Godel published his "incompleteness theorem" showing that there is no such system. In the late 1930s, Turing discovered a similar result, that there are problems that cannot be solved by any mechanical procedure. The importance of logic was not only its deep insight into the limits of automatic calculation, but also its focusing on the possibility that strings of symbols, perhaps encoded as numbers, can be interpreted both as data and as programs.

This insight is the key idea that distinguishes the stored program computer from calculating machines. The steps of the algorithm are represented as binary codes and stored in the memory for later decoding and execution by the processor. The binary code can be derived mechanically from a higher-level symbolic form, the programming language.

It is the explicit, and intricate, intertwining of the ancient threads of calculation and logical symbol manipulation that marks the birth of the discipline of computer science.

Computer science has grown from infancy in the 1940s to a broad discipline in the 1980s. The box below traces this development, showing times at which new subfields made the transition from poorly understood sets of techniques to well understood sets of core principles. The dates shown in the box are, of course, approximate. They represent my estimates of when these areas were included in the required courses in a significant number of computer science departments. Artificial intelligence, which has been an active research area since the early days of computer science, is now making the transition from elective to required status at many universities.

The eleven areas are by no means mutually exclusive. Each has its own theoretical component; most have devised specialized programming languages as notation for algorithms and data structures; most implementations are on machines with operating systems connected to networks; most deal with problems having components that can execute in parallel. Some subdisciplines, such as software engineering, embrace all eleven areas.

The following paragraphs outline the principal content of the eleven areas, listing the fundamental questions and the major accomplishments of each.

| The Evolution of Computer Science | |
|---|---|
| Theory | 1940 |
| Numerical computation | 1945 |
| Architecture | 1950 |
| Programming languages & methodology | 1960 |
| Algorithms & data structures | 1968 |
| Operating systems | 1971 |
| Networks | 1975 |
| Human interface | 1978 |
| Database systems | 1980 |
| Concurrent computation | 1982 |
| Artificial intelligence | 1986 (?) |

THEORY. This area deals with the basic mathematics underlying computation. The fundamental questions are: What problems can machines solve? What are optimal algorithms for given classes of problems? What is the intrinsic best and worst case performance of given classes of machines for given classes of problems? What problems are equivalent to each other in computational difficulty? The major accomplishments are:

1. Computability theory, which defines what machines can and cannot do. Branches include automata and formal language theory.

2. Complexity theory, which tells how to measure the time and space requirements of computable functions. This theory relates a problem's size with the best- or worst-case performance of algorithms that solve that problem.

3. Classification of problems into complexity classes, such as those solvable deterministically in polynomially-bounded time (P-problems) and those solvable nondeterministically in polynomially-bounded time (NP-problems).

4. Automatic theorem proving.

NUMERICAL COMPUTATION. This area deals with general methods of efficiently and accurately solving equations resulting from mathematical models of systems. The fundamental questions are: How can we accurately approximate continuous or infinite processes by finite discrete processes? How do we cope with the errors arising from these approximations? How rapidly can a given class of equations be solved for a given level of accuracy? How can symbolic manipulations on equations, such as integration, differentiation, or reduction to minimal terms, be carried out? How can the answers to these questions be incorporated into efficient, reliable, high-quality mathematical software packages? The major accomplishments are:

1. Theories of stability of methods and error propagation resulting from finite and discrete representations – in particular, backward error analysis.

2. Fast algorithms for certain problems such as the fast Fourier transform and solution of Poisson's equation. Extensive assessment of algorithms for accuracy and efficiency.

3. The finite element model for a large class of problems specifiable by regular meshes and boundary values. Associated iterative methods and convergence theory. Automatic grid refinement during numerical integration.

4. Mathematical software packages for handling general problems involving matrices, ordinary differential equations, and statistics; and less general problems involving partial differential equations, optimizations, and nonlinear equations.

5. Symbolic manipulators capable of powerful and nonobvious reductions, differentiations, and integrations of expressions.

**ARCHITECTURE.** This area deals with methods of organizing many hardware and software components into efficient, reliable systems. The fundamental questions are: What are best methods of implementing processing, memory, and communication functions in a machine? How do we build large computational systems in such a way that we can convincingly demonstrate that they work as intended despite various types of errors and failures? The major accomplishments are:

1. Finite state machine theory and Boolean circuit algebra, which relate hardware function to structure.

2. The so-called von Neumann machine, which is the single-instruction sequence stored program computer.

3. Hardware units for fast arithmetic.

4. Efficient methods of encoding and storing information in various media.

5. Theory of reliable computing systems, including redundant components, reconfiguration, diagnostics, and testing.

6. Methods of synthesizing large complex systems from basic components.

7. Prototypes of multiprocessor machines capable of supporting hundreds or thousands of simultaneously executing processors.

8. Microelectronic circuit technology and computer aided design of very large scale integrated (VLSI) circuits.

**PROGRAMMING LANGUAGES AND METHODOLOGY.** This area deals with notations for expressing algorithms and data, with efficient translations from high level languages into machine codes, and with methods of efficiently constructing correct programs. The fundamental questions are: What are the basic data types and operations that arise in various classes of problems and how should they be represented? What are the basic methods of controlling the execution of a computation? How can syntactic descriptions of language be used to construct efficient compilers and optimal code generators? What methods should be used to aid in the process of proving that a program performs its intended function? The major accomplishments are:

1. Procedure-oriented programming languages such as Cobol, Fortran, Algol, Pascal, or Ada. Functional languages such as APL, Lisp, Prolog, and VAL. Object-manipulating languages such as Smalltalk or CLU.

2. Codification of basic concepts of programming languages such as basic data types (e.g., scalars, arrays, records, strings) and control structures (e.g., sequencing, iteration, selection, subroutines, recursion).

3. Theory of compiling and code generation and its application in real compilers.

4. Verification, which deals with establishing that a program's functional specifications are satisfied by its implementation.

5. Syntax-directed editors that monitor program construction and alert the user to potential errors.

**ALGORITHMS & DATA STRUCTURES.** This area deals with specific classes of problems and their efficient solutions. The fundamental questions are: For given classes of problems, what are the best algorithms? How much storage and time do they require? What is the tradeoff between space and time? What is the worst case of the best algorithms? How well do algorithms behave on average? How general are algorithms – i.e., what classes of problems can be dealt with by similar methods? The major accomplishments are:

1. Identification of good and bad algorithms for important classes of problems such as searching, sorting, random-number generation, and textual pattern matching.

2. Identification of general methods applicable across many classes of problems, such as storage of information in tables or lists, graph algorithms, or tree algorithms.

3. Categorizing the effects of data structure on time and space requirements of programs for various classes of problems.

**OPERATING SYSTEMS.** This area deals with the control mechanisms that allow multiple resources to be efficiently coordinated in the execution of programs. The fundamental questions are: At each time scale in the operation of a computer system, what are the visible objects and permissible operations on them? For each class of resource (objects visible at some level), what is a minimal set of operations that permit their effective use? How can interfaces be organized so that users deal only with abstract versions of resources and not with physical details of hardware? What are effective control strategies for job scheduling, memory management, communications, access to software resources, communication among concurrent tasks, reliability, security, and the like? What are the principles by which systems can be extended in function by repeated application of a small number of construction rules?. The major accomplishments are:

1. Prototypes of timesharing systems, interrupt systems, automatic storage allocation, schedulers, and file systems that served as the bases of major commercial systems. Libraries of utilities such as text editors, document formatters, compilers, linkers, and device drivers.

2. Powerful, hierarchical abstraction principles that permit users to operate on idealized versions of resources without concern for physical detail -- for example, processes instead of processors, files instead of disks, data streams instead of program input/output.

3. Theories of process management including reliable interprocess synchronization, communication, and deadlock control.

4. Theories of memory management including optimal swapping policies for virtual memory, file access methods, and secondary storage optimization.

5. Hierarchies of directories.

6. Theories of job scheduling, queueing network modeling, and other forms of performance modeling.

7. Models of access control for files owned by specific users.

8. High level command interfaces that permit users to easily express computations consisting of several components selected from among the files in a system. This includes interactive "windows," command "menus," and pointers such as the "mouse".

**NETWORKS.** This area deals with the organization of systems comprising interconnected computers. The fundamental questions are: What are the most efficient methods of error checking and correction? Of reliably exchanging information across various media (e.g., telephone lines, microwaves, laser optics)? Of mediating contention for shared channels? What strategies (protocols) should be used for connecting computers across long distances? Short distances? How can the fact that a system is made of components connected by networks be hidden from users who do not wish to see that level of detail? The major accomplishments are:

1. Prototyes for long-haul, computer-to-computer communication networks that served as the bases for commercial networks.

2. Local networks for high speed connections among close computers, such as Ethernet, Pronet, or token-ring nets.

3. Protocols that allow computers to establish and maintain connections across unreliable networks.

4. Protocols that mediate high-speed contention on shared or broadcast channels.

5. Cryptographic protocols that permit secure authentication and secret communication.

6. Structural principles for operating systems that allow hiding of the network from those who do not wish to see it.

**HUMAN INTERFACE.** This area deals with the transfer of information between humans and machines via various human senses and motor skills. The fundamental questions are: What are efficient methods of representing objects and automatically creating pictures for viewing? What are efficient methods for receiving input or presenting output? How can the risk of misperception and subsequent human error be minimized? The major accomplishments are:

1. Core graphics systems for representing objects, for displaying them efficiently, and for creating displays that rotate, translate, pan, and zoom in real time. This includes a wide range of algorithms for constructing

pictures from basic components, smoothing, shading, and removing hidden lines.

2. Interactive methods for computer aided design.

3. Advanced forms of input and output such as optical readers, light pens, touch sensitive pads, and the "mouse" pointer.

4. Psychological studies leading to modes of interaction that reduce human error and increase human efficiency.

**DATABASE SYSTEMS.** This area deals with the organization of large sets of data for efficient queries. The fundamental questions are: What basic models should be used to represent data elements and relations among them? What operations are used to store, locate, retrieve, and match data? How can these operations most efficiently be expressed in language forms? How can high-level descriptions of queries be translated into efficient codes for sifting through the database? What machine architectures lead to the fastest retrievals? How can the data be protected against unauthorized access, disclosure, or destruction? How can large databases be protected from inconsistencies generated by simultaneous access, especially when the data is distributed among many machines? The major accomplishments are:

1. Major models for representing large data sets and relations among the data elements, including the relational, hierarchical, and network models. Special representations of files for fast retrieval, such as inverted trees and associative stores.

2. Design principles for locking records when they can be simultaneously accessed by many users.

3. Design principles for maintaining consistency among multiple copies of data stored on different machines of a network.

4. Principles for preventing unauthorized disclosure or alteration of information in the database, including protection against statistical inference in real-time query systems.

5. High performance database machines.

**CONCURRENT COMPUTATION.** This area deals with the organization of computations that require many processing elements working concurrently. The fundamental questions are: What are the basic models of concurrent computation? What classes of problems are most effectively served by each model? What types of machines are most suited for efficient implementation of programs in each model? What high level visual tools should be provided so that massively parallel computations can be expressed quickly and correctly? How can the large numbers of resources required in such computations be efficiently managed? The major accomplishments are:

1. General models for parallel computation such as tree machines, mesh array machines, dataflow machines, and communicating sequential processes; new programming languages for these machines.
2. Development of parallel algorithms for important problem classes on these machines; methods of partitioning problems into parts that can be executed concurrently; division of parallel algorithms into time and space complexity classes.
3. Interactive aids for programming and debugging parallel computations.

**ARTIFICIAL INTELLIGENCE.** This area deals with the simulation of intelligence. The fundamental questions are: What is intelligence? What basic models of intelligence are there and how do we build machines that simulate them? To what extent is intelligence described by rule evaluation and what is the ultimate performance of machines that simulate intelligence by evaluating rules? To what extent is intelligence unpredictable and can this be modeled by randomness in the machine? The major accomplishments are:

1. Theories of cognition and thought expressed in terms that could be realized by computer.
2. Efficient methods of knowledge representation and searching through knowledge bases.
3. Powerful software systems for logic programming, theorem proving, and rule evaluation.
4. Special applications such as robotics, image processing, vision, and speech recognition.
5. Expert systems based on rule evaluation for simulating expert human behavior in a few narrow domains.

Computer science includes in one discipline its own theory, experimental method, and engineering. This contrasts with most physical sciences, which are separate from the engineering disciplines that apply their findings -- as for example, in chemistry and chemical engineering. I do not think the science and the engineering can be separated within computer science because of the fundamental emphasis on efficiency. But I do believe that the discipline will endure because the fundamental question -- "What can be automated?" -- will not soon be answered.

## *Reference*

1. Readers interested in a detailed treatment of the subjects covered here are invited to examine the report of the NSF Computer science and Engineering Research Study (COSERS), *What Can be Automated?*, ed. B. Arden, 1980, MIT Press.

# Computer Networks

*American Scientist 73*
No. 2 (Mar-Apr 1985)

*Computer network software is organized into a series of layers that successively hide more and more of the detail of signal transmission over single links, signal transmission over multiple-link paths, error and congestion control, reliable computer-computer connections, and filtering data as it moves between user programs and the network.*

In mid 1984, the National Science Foundation (NSF) formed the Office of Advanced Scientific Computation to establish national supercomputer centers and make them accessible to the entire scientific community through a network to be called Sciencenet. This project has drawn scientists from many disciplines into potentially bewildering discussions about computer networks. In this column I will discuss the basic principles of networks. For further reading that will introduce you to this fascinating subject, I suggest the materials by Tanenbaum at the end of the article.

A computer network is a collection of computers, called "hosts", that can communicate with one another. A host can be a large supercomputer, a time-shared minicomputer, or a personal workstation. Ordinary terminals are not considered hosts.

There are two types of network, local and long-haul. A local network is used to connect computers in the same building or in neighboring buildings. It is built of special cables and interfaces that achieve high speed by taking advantage of the low error rates possible over short distances. Local networks capable of transmission rates of 50 megabits per second (Mbps) are available commercially. A long-haul network is used to connect computers over long distances. It is typically built from telephone or satellite links. The transmission rates are much lower, ranging from 1.5 Mbps on satellite links, to 56 kilobits per second (Kbps or "kilobaud") with the best special-purpose modems over leased telephone lines, to 9.6 Kbps with the best modems over dial-up lines, to 1.2 Kbps with inexpensive modems over dial-up lines. Whereas local networks are normally operated by the same organization who own the computers, long-haul networks are normally operated by outside organizations, the common carriers.

Two modes of data communication are generally used, broadcast and packet-switched. In broadcast mode, the entire network is treated as a single channel shared by all the hosts; the bandwidth can be very high but interference (called "collisions") can be a problem if each host has a high need for the network. The broadcast mode is common in local networks. (Some satellite links also operate in broadcast mode.) In packet-switched mode, the incoming data stream is broken into chunks, which are stored in packets that also contain the address of the destination host; the network relays the packets through a series of switch computers, often called IMPs (for Interface Message Processors), en route to their destinations. This mode is used to efficiently share many computer-computer conversations over common links. The packet-switched mode, also called store-and-forward mode, is common in long-haul networks.

While the concept of connecting computers together has long fascinated the writers of science fiction, the first long-haul computer network became reality in the early 1970s. This was the ARPANET, built under contract to the Defense Advanced Research Projects Agency (DARPA). The ARPANET continues to provide high-grade service today, but its access is limited to hosts under contract to the Department of Defense, which includes about twenty universities, and the cost is high ($130K per IMP in 1984). Many spin-offs of the ARPANET technology now exist, such as the NSF-initiated computer science research net (CSNET) and commercial networks such as GTE Telenet, Unidata, and CompuServe. Local networks began to appear in the middle 1970s, with such notable examples as Ethernet$^{TM}$ of the Xerox Corporation and the University of Cambridge Ring. Local networks are now a major commercial enterprise.

Why connect computers? One reason is to enable the use of remote resources – for example, remote job entry to a supercomputer, remote use of a powerful graphics facility, or remote use of a chip fabrication facility. A second reason is to enable sharing of programs and data – for example, one institution can make its special codes and data libraries available to the rest of the community and thereby space other institutions needless duplication of effort. A third reason is to encourage collaboration among the users of the network. The experience of the ARPANET, which is being reconfirmed in CSNET and other networks, is that collaboration, often envisioned as the least of the three reasons, is in fact the most important. The ability of a network to knit together the members of a sprawling community has proved to be the most powerful way of fostering scientific advancement yet discovered.

How are networks built? Despite a wide variety of implementations and a massive amount of technical detail, a set of design principles has emerged. These principles have been captured in a model called Open Systems Interconnections promulgated by the International Standards Organization in 1980. The model organizes the functions of a network into a hierarchy according to their characteristic time scales and levels of abstraction. Each layer builds on, and adds functions to, the layers below. Layers from, highest to lowest, are:

| | | |
|---|---|---|
| 7 | Applications | User programs such as file transfer, remote job entry, and electronic mail |
| 6 | Presentation | Filter user data as it moves between applications programs and the network |
| 5 | Session | Transmit information reliably between a process on one host and a process on another |
| 4 | Transport | Transmit information reliably over a data path from one host to another |
| 3 | Network | Find and control multi-IMP routes from one host to another |
| 2 | Data Link | Transmit data reliably in packets over a single data link |
| 1 | Physical | Transmit electrical signals over a cable or other channel |

I will discuss these layers in the sections following, beginning with Layer 1 and working up.

The software (and hardware) implementing these layers must be present in each host of the network. The software is designed so that Layer $i$ on one host can interact with Layer $i$ on another host as if the lower layers did not exist. To implement this, the data being transmitted from Layer $i$ on one host are passed down through the software at Layers $i-1,...,1$, being transformed by each as they pass through. The data are transmitted to the physical layer of the destination host, which passes them up through the software at Layers $1,...,i-1$ where they reappear at Layer $i$.

The algorithms implemented by the various software layers are called network protocols. Network protocols are sometimes part of the operating system. Since an IMP is merely a relay computer, the protocol software contained in it need only span Layers 1, 2, and 3. Protocol software tends to have arcane acronyms left over from the deliberations of the standards committees. One of the physical layer protocols approved is designated "X.21". One of the protocols for Layers 1-3 is designated "X.25". The software for the ARPANET covers layers 1-5 and is designated "TCP/IP" (for transport control protocol and internet protocol). Although these protocols span the same layers in the model, they may be incompatible; for example, the strategy used in X.25 for layers 1-3 is different from the strategy used in TCP/IP for the same layers.

**THE PHYSICAL LAYER.** This is the hardware level at which the actual transmission of a raw bit stream from one station to another takes place. All electrical and mechanical aspects of data communication are handled at this level. The designers must answer such questions as how to represent bits (0 or 1) as signals, whether half duplex (unidirectional communication) or full duplex (bidirectional communication) will be used, what the pin configurations on the connectors will be, and what type of network the host will be part of.

Most physical networks employ analog signalling. Thus, a modem is needed to convert between the computer's digital data and the telephone line's voice bands. The international organization for standards in telephony has proposed an all digital protocol, X.21, for connecting a digital host to a digital physical network.

Local networks present interesting challenges at the physical layer. These networks are typically operated in broadcast mode. Each host listens continuously and receives data by picking out messages whose headers are addressed to it. Special control schemes select which of several contending hosts will actually get to use the network – simultaneous transmissions will jam one another.

In an Ethernet$^{TM}$, for example, hosts are attached to a single coaxial cable. To send data, a host waits until no signal is present on the cable, then begins transmitting. If it discovers that it is being jammed ("collided with") by another host, it stops, waits a random amount of time, then retries. This strategy is embodied as a protocol called Carrier Sense Multiple Access with Collision Detection (CSMA/CD). In a "token ring" local network, the hosts are connected in a circle. A special signal pattern, called the control token, is passed around the ring until it comes to a host having data to transmit. The host seizes the token and transmits the data; when done, it transmits the control token. Ethernets capable of 10 Mpbs and token rings capable of 50 Mbps are commercially available.

The user of Layer 1 can be sure that a given string of bits will be encoded and transmitted, but cannot be sure the data have passed successfully over the data link. Errors are detected and fixed by the next higher level.

**THE DATA LINK LAYER.** This level provides reliable physical links between adjacent hosts or IMPs. The basic strategy is to divide the data into chunks, called frames, and then to embed each frame in a packet for transmission. A packet contains additional information such as destination address, a sequence number, and redundant bits called a checksum for detecting errors. Packets are typically in the range from 10 to 1000 bytes long. (A byte is an 8-bit code for a character.) The sending host's data link layer transmits the packets in sequence and waits for acknowledgements from the receiving host; it retransmits packets for which acknowledgements are not received within a time limit. The receiving host's data link layer attempts to collect a complete sequence of correct packets in its buffer. A packet is correct if the checksum in the packet matches a checksum computed for the remaining information in the packet.

Because of the finite size of its buffers, the receiving host may not be able to accept packets as fast as the sender can transmit them. For this reason, data link protocols include a mechanism, called flow control, whereby the sender is shut off whenever the number of unacknowledged packets exceeds a limit. A standard protocol embodying these ideas is called High Level Data Link Control

(HDLC).

The user of the data link layer can be sure that a given string of bits will be transmitted and acknowledged correctly over a given link, but cannot send information over paths comprising multiple links. Multiple-link paths are set up and managed by the next higher level.

**THE NETWORK LAYER.** This level provides multilink paths from host to host using one or more IMPs as relays along the way. The basic strategy is to place a "routing table" in each IMP that tells what link to use when forwarding a packet addressed to a given host. A packet received by the data link layer on an IMP is passed up to the network layer, which then determines which outgoing link to use and generates a new transmit request on the data link layer. Routing tables can be dynamically updated so that packets are sent out along links with the least congestion; the information about queue lengths within an IMP can be sent periodically to neighboring IMPs in special control packets. Two basic strategies for the network layer protocol are in use, virtual circuits and datagrams.

A virtual circuit is a predetermined path (series of links and IMPs) over which all packets in a conversation between a given pair of hosts flow. It is established when the caller sends a special "call request" packet to the receiver. The path traced by the call request packet is remembered in the IMP routing tables along the way. The receiving host returns a "call acknowledge" packet. Both hosts embed a virtual circuit number in each data packet so that those packets can efficiently trace the same path and can be efficiently acknowledged. The term virtual circuit suggests the simulation of a telephone circuit set up through a series of relays when a call is made. The advantage of virtual circuits is that the exchange of data packets can be very efficient once the circuit is open because virtual circuit number fields in packets are short and the required buffer space in the IMPs along the way has been reserved in advance; the disadvantage is that a lot of IMP memory can be wasted remembering virtual circuits over which there is little traffic. The international body for standards in telephony has established a protocol called X.25 embodying these ideas. Networks based on the X.25 protocol are common in Europe and are provided by some carriers in the U.S., notably GTE Telenet, Unidata, and CompuServe.

A datagram is a packet sent independently of all other packets, past or future, belonging to the same conversation. It must contain the full address of the recipient. Any message that can be completely embedded in a single packet is much cheaper to send in a datagram than in a virtual circuit network. The advantage of datagrams is simplicity in the Layer 3 protocol; the disadvantage is that the software above Layer 3 has a greater responsibility to check that all components of a message have been received and properly acknowledged. A datagram oriented protocol, called Internet Protocol (IP), is used in the ARPANET.

The user of the network layer can be sure that packets that must traverse multiple links can be guided to their destinations, but cannot be sure they all arrive in order or intact. With datagrams, for example, packets can arrive out of order because they followed different paths through the IMPs. With virtual circuits, the failure of an IMP can break the circuit and lose packets buffered in that IMP. The next level up overcomes these difficulties.

**THE TRANSPORT LAYER.** This layer provides reliable multilink paths between pairs of hosts. The software includes tests to verify that circuits remain open or datagrams are eventually acknowledged. The interface contains these commands:

```
con-id = OPEN(localport, remoteport)
con-id = LISTEN( )
CLOSE(con-id)
SEND(con-id, address, length)
RECEIVE(con-id, address, length)
```

The OPEN command attempts to establish a connection between a local port on the calling host and a given remote port on the called host and returns a connection number if successful. The LISTEN command waits for an incoming call (generated by an OPEN command on another host) and returns a connection identification number. The CLOSE command breaks the connection. The SEND command sends a series of bytes of given length over an open connection; the beginning of the series is at the given memory address. The RECEIVE command waits until the connection contains the requested length byte-sequence, then stores it at the given address.

The ARPANET uses network software called Transport Control Protocol (TCP) and offers datagram service over leased telephone lines fitted with 56Kbps modems. (Overhead, from congestion in the IMPs and from transmitting identifying information along with the messages, causes the effective average file transfer rate to be much lower, typically under 20 Kbps.) Each IMP is connected to at least two other IMPs so that the failure of one IMP or telephone line will not isolate any part of the network. The underlying protocol for delivering datagrams that may possibly be addressed to networks other than the ARPANET is called Internet Protocol (IP). The combination is referred to as TCP/IP.

The user of the Transport Layer can be sure that messages will be reliably delivered to remote hosts irrespective of the state of the network, number of links on the path, datagram or virtual-circuit service, or number of operational IMPs. Moreover, the details of the network technology are completely hidden in the sense that the same interface can be used with networks ranging in speed from telephone lines to satellite links. Connections between user processes (running programs) and ports on a host are managed by the next level.

**THE SESSION LAYER.** This layer establishes and manages reliable connections between pairs of processes (running programs) on different hosts. It is a minor extension of the transport layer and performs such additional useful functions as allowing symbolic names (i.e., character strings rather than numbers) to be used in the calls on the OPEN, CLOSE, SEND, and RECEIVE commands, or matching responses from remote processes with multiple outstanding requests to those processes. In most systems today there is little distinction between the Transport and Session Layers and there is some question whether the model should maintain such a distinction.

**THE PRESENTATION LAYER.** This layer filters (transforms) data in certain useful ways as it moves between user programs and the network. Some of functions are:

1. Encryption Protocol. The contents of messages are encrypted on transmission and decrypted on receipt. The distribution of keys is the most challenging aspect of network encryption.
2. Text Compression Protocol. A considerable amount of redundancy exists in text data. Simple text compression algorithms reduce text files by 50% before transmission. This can signficantly increase the effective bandwidth of the network.
3. Virtual Terminal Protocol. This software allows users to write applications programs that will work correctly with any terminal connected across the network to the program. The idea is to define a hypothetical terminal and require programmers to deal only with it. The protocol software converts between the commands of the actual terminal and the corresponding commands on the virtual terminal.

**THE APPLICATIONS LAYER.** This is a catch-all layer in which reside all the user programs that may require an interaction with the network. The most prominent examples are electronic mail programs, remote job entry programs, and file transfer programs.

As more networks are built, the interest in connecting them together increases. A computer connecting two networks is called an internet gateway. Gateway computers must know about the protocols used on both networks and must convert packets from one format to the other. Sometimes the gateways must enforce access controls, rules that constrain which connections are allowable. Because the Open Systems Interconnection model does not discuss gateways, I have not attempted to fit them into the hierarchy in this discussion. They are nonetheless important in the supernetworks that will be built in the future.

The above description has focused on the internal structure of networks. Most of this structure is hidden from users who do not wish to see it. Very little has been said about how networks appear to their users. That will be the subject of the next column.

## *References*

Tanenbaum, A. S. 1981. "Network protocols." ACM *Computing Surveys 13* (4), 453-489.

_______ 1981. *Computer Networks.* Prentice-Hall.

# Supernetworks

*American Scientist 73*
No. 3 (May-Jun 1985)

***Supernetworks such as Sciencenet will be formed by connecting many different networks with gateways. The resulting network must appear to be a coherent system providing access to a wide range of community resources through a common user interface.***

In the March-April issue, I discussed the internal structure of computer networks, emphasizing how the protocol software can be built as a series of layers. Most of this structure is hidden from the users of the network. What should the users see?

Preliminary answers can be given in the context of the Advanced Scientific Computing Initiative, the new project of The National Science Foundation (NSF) to make national supercomputer centers accessible to the entire scientific community. This project includes a network which I referred to as Sciencenet in the pevious column. It turns out that a private company is using the name Sciencenet, and so a new name must be found. Not knowing what the name will be, I will coin the term "NSF-net" for the remainder of this column. NSF-net will be a network of networks connected by gateways, a sprawling, complex, heterogeneous system. Yet it must provide simple, uniform access to community resources. Properly designed, such a system would be a powerful tool for science – a "supernetwork."

NSF has stated that one of the most important goals of its Advanced Scientific Computing Initiative is to promote cooperation and sharing of advanced computational resources among all members of the scientific and engineering community. NSF-net is charged with developing or promulgating standard transport, gateway, and application-level protocols toward this goal. What might this supernetwork do for you as a user? A well-designed NSF-net could:

1. Allow you to interact regularly with research colleagues around the world without having to know exactly where they are or what kinds of computers they use.

2. Allow any program, database, service, or facility in the network to be registered as a network-wide resource. (In particular, allow you as an individual to share any resources you create with the entire scientific community.)
3. Allow you to quickly find the names of registered resources when all you know is their general function. And then allow access to these resources without knowing exactly where they are or what computers they use.
4. Allow access to resources by the same interface irrespective of whether they are local or remote.
5. Allow you to construct programs that not only can use resources around the network but will not malfunction if those resources are moved.

Five principles that can jointly yield such a design are described in the sections below.

PEOPLE-ORIENTATION. The network's administration should actively encourage interaction among scientists. This is so for at least two reasons. First, collaboration and sharing are the most powerful methods known for advancing scientific knowledge. Second, building on existing resources is far more productive than creating one's own versions. People-orientation has many implications for the design of network software at all levels. For example:

1. A meaningful set of external names for resources is essential in addition to internal addresses that are easily processed by software and hardware.
2. Because names for resources will be embedded in programs, files, and directories, names must be location independent. Then, if the physical location of a resource changes, none of the programs, files, or directories that refer to it need be edited or recompiled in order to continue working correctly.
3. The network must provide a variety of interactive "help servers," which are advanced database systems that can respond to natural-language queries about network resources. Help servers will enable users to find and use existing resources rather than building new versions.
4. The primary mechanism of interaction among users will be electronic mail, which may eventually include voice and video components.
5. A good administrative structure is important to answer user questions, help new institutions come on line, support mailing lists, manage bulletin boards, develop new services, set standards, negotiate rates with public carriers, repair faulty software, obtain community input on network policies and operations, conduct measurements, and undertake experiments for improvements.

**RESOURCE GENERALITY.** The network should place no prior constraints on what programs, data, systems, or facilities can become network resources.

The first resources on NSF-net will be supercomputer centers. This is because NSF's first priority is to provide advanced computational power to the scientific community. In time, however, the community will begin to accumulate new knowledge from results obtained by advanced computation. The new knowledge should become accessible and usable by the community. This can be accomplished by allowing (and encouraging) users to develop new computational resources and attach them to the network. Examples include:

| | | |
|---|---|---|
| Program libraries | – | Software, for instance, for image processing, statistical analyses, or graphics support. |
| Data Libraries | – | Raw or condensed data from experiments or satellite instruments, partially processed images, catalogs of new astronomical objects, or catalogs of chemical compounds. |
| Services | – | Software distribution services, information services that aid in locating people or data of special kinds (help servers), and directories of users or other services. |
| Facilities | – | For example, a center offering special processing and expertise in a discipline (e.g., NASA Ames's Numerical Aerodynamic Simulator), a real-time satellite data collector, a 3-D graphics processor, or an astronomical laboratory delivering real-time data. |

**LOCATION INDEPENDENT NAMING.** The network should provide uniform access to its resources without requiring users to know the physical locations of resources or other users. Location independence, also called "location transparency" in the jargon, is the property that external names – the character strings that are meaningful to users – are interpretable by the network no matter what the physical addresses of the objects or persons denoted. This principle is enforced partly by network administration and partly by designing software to distinguish name from address. Users of networks with this property can be assured that commands and programs will continue to work even if files or users should move.

Network nameservers play prominent roles in location independence. These databases record the correspondences between external names and internal addresses. Nameservers containing directories of users are maintained at the network administrative centers of both ARPANET and CSNET. Future nameservers should be more powerful – advanced database systems that can be queried by keywords as well as resource names. These "help servers" will allow users to locate objects, knowing only their functions.

The problems arising when the network forces users to deal with addresses directly can be serious. Consider a scenario. A Stanford scientist constructs a program to read and interpret DNA sequences stored in a database called DNA-Traces. This program is called DNA-Reader and is widely distributed in the scientific community. Under the current ARPANET addressing conventions the database receives the network name DNA-Traces@Stanford (read as "DNA-Traces at Stanford") and the program DNA-Reader@Stanford. The DNA-Reader program contains the string "DNA-Traces@Stanford" at the point where it interrogates the database. A user anywhere in the network can obtain a copy of DNA-Reader from Stanford. When any copy of DNA-Reader is run, it can use this string to address the database.

Eventually Stanford creates the DNA Institute and moves the database to the new location, thereby effectively renaming it DNA-Traces@DNA-Institute. Suddenly hundreds of copies of DNA-Reader stop working because the address "DNA-Traces@Stanford" implanted in them is invalid. (Few users would understand the failure; fewer still would be able to fix the program.) Before new copies of DNA-Reader can be distributed, a student named Donald N. Aldoon at Stanford installs a personal file containing event traces from a simulated disk storage system – naming it DNA-Traces. Suddenly all those DNA-Reader programs seemed to work again ....

This scenario illustrates two problems caused by location-dependent addresses. First, all copies of software (or databases) referring to a resource will cease to work if they refer to a resource that is moved. Second, these programs will incorrectly refer to the wrong resource if the original name of the moved resource is reused. These problems are not simply inconveniences. They are costly because they make software resources unreliable and the malfunctioning software may not be detected.

Location-independent names are a necessity for reliable software in a resource-sharing network. Here's how the DNA scenario would work. The Stanford scientist would register the program and database with the network nameserver; suppose the strings "DNA-Reader" and "DNA-Traces" are approved and are unique in the network. The scientist assigns corresponding local names unique only within the Stanford computer; suppose these are, respectively, "DNAR001" and "DNAT001". The scientist informs the nameserver of these local names and the nameserver installs the entries

DNA-Reader = DNAR001@Stanford
DNA-Traces = DNAT001@Stanford

Later, when the database is moved, its nameserver entry can be corrected to read

$$\begin{aligned} \text{DNA-Reader} &= \text{DNAR001@DNA-Institute} \\ \text{DNA-Traces} &= \text{DNAT001@DNA-Institute} \end{aligned}$$

All programs and databases that refer to the DNA-Reader and the DNA-Traces database do so by their network names, which never change. Every network command that receives a location-independent resource name as a parameter works like this: it interrogates the nameserver to obtain the current network address corresponding to the name, which it then passes to the protocol software for access to the resource. The interaction with the nameserver to translate the name is done automatically and is not visible to the user.

Now anyone wishing to obtain a copy of the DNA-Reader program can do so simply by a command like OBTAIN-COPY(DNA-Reader), which would work before and after the program is moved to the DNA Institute. Moreover, none of the copies of the program would need revision because they refer to the database only by its unique name which does not change when the database moves.

Location-independent naming may at first appear slower than location-dependent addressing because of the interaction with the nameserver. Almost all this interaction can be eliminated by maintaining in each host a small database, called a cache, containing copies of the most recently used entries in the nameserver. (Safeguards are needed to ensure that cache entries become invalid if a resource moves.)

Location-independent addressing also allows users to interact without having to know exactly where they are. In the current ARPANET, for example, someone wishing to send me mail must issue a command like MAIL(pjd@RIACS), where "pjd" is my local login identifier and "RIACS" the acronym for my institute. If I change hosts, those wishing to send mail to me encounter the same problem as the DNA-Reader when the DNA-Traces database moved: the MAIL command malfunctions. If I am registered with the nameserver under the unique name, say, PJD1, the command MAIL(PJD1) will work correctly no matter where my mailbox is, as long as I keep the nameserver informed of my current location.

Network names may be artificial or clumsy -- for example, I may not be allowed to register under my initials (PJD) because some other user has already used those initials. Users prefer short, descriptive names having more meaning, and so many systems allow each user to define an "alias table" that records nicknames for resources and users. Now the addressing protocol becomes, first, translate alias to network name; then, translate network name to network address. For example, one user can say "set alias peter = PJD1" after finding my unique name from the nameserver; thereafter commands MAIL(peter) by him will work correctly. Meanwhile, another user can say "set alias riacs-director = PJD1" and for him MAIL(riacs-director) will work correctly. The alias table is not the same as the addressing cache mentioned earlier: the alias table is established and maintained individually by each user and does not interact with the nameserver.

**CONNECTION SIMPLICITY.** The same method should be used throughout the network for opening reliable connections between processes; moreover, the method of sending data over an open connection should be independent of the physical path, bandwidth or communications technology used. Factors concerning physical details such as network types, protocols, bandwidths, transmission modes, media, routings, gateways, addressing schemes, and operating systems need not be visible to anyone except the few experts who wish to see them.

The most effective way of hiding these details is to regard a connection merely as a transmission path for a few selected types of data between two nodes of the network. The two ends of a connection are called sockets. Devices, files, and executing programs can be connected interchangeably to sockets. The type of data passed over a connection cannot be completely hidden because it affects transmission priorities and error controls on packets. The major types are bulk bit streams, video frame updates, and voice.

The network protocol software is responsible to set up the path over one or more networks that physically connect the two hosts between which connection is desired. The user will not be required to be aware of the details.

Scrupulous application of the connection-simplicity principle can hide a surprising amount of detail, but not all. It can certainly hide the working details of commands to open, close, send, or receive – but it cannot hide measurable properties such as bandwidth or response time.

**STANDARD EXTENDABLE INTERFACES.** The network should provide higher-level functions as standard extensions to the commands. The essential commands are:

Open a connection with another node.
Listen for an incoming call.
Close a connection.
Attach a file, device, or program to a socket.
Detach a file, device, or program from a socket.
Send data into an open connection.
Receive data from an open connection.
Find out the status of a connection.

An interface this simple can provide most of the function we expect of a network. The generality of this interface can be illustrated with a session of remote interactive use of the DNA-Reader program. The user has a file called DNA-options that is input to the DNA-Reader and controls the type of reading performed. The user wishes to watch the evolving interpretations on a local graphics display. In the absence of a standard interface for remote program execution, the user would have to perform these steps:

1. Open two connections, for both directions of communication with the remote computer containing the DNA-Reader. This pair of connections will serve as the control channel with the remote computer. By default, the remote operating system will attach the input and output ports of a login program to them.
2. Attach the local keyboard and display to this connection pair.
3. Type the commands needed to log in to the operating system on the remote computer.
4. Open two more connections with the remote computer, to be used as the input-output channel with the remote DNA-Reader process. Type the commands needed to instruct the remote operating system to spawn a process containing the DNA-Reader program and connect its input and output to this connection pair. Locally, attach the DNA-options file to the outgoing connection and the interactive graphics display to the incoming connection.
5. Using the control channel, issue further commands as necessary to control the remote process.
6. When the remote process has completed, close down all the above connections.

The above scenario contains at least twelve calls on commands in the connection interface. It also contains standard command sequences for login and job activation. The overall sequence of steps is likely to be so common that a command script can be written embodying them. The user then need execute only the command script with a few parameters to describe the remote program and the local sources of input and output. The command script might have the form

REMOTE-EXECUTE(host, program, input, output).

The scenario above could be called into action with the single call

REMOTE-EXECUTE(Stanford, DNA-Reader, DNA-options, Display).

Note how a command script can hide the network command interface behind a simpler, higher-level interface. It is in the best interest of the network to specify standards for high-level interfaces of common functions such as remote execution.

With a sufficiently powerful local operating system, most of the above steps can be executed implicitly by the local command interpreter, which would hide even more of the network interface. For example, a simple extension to the UNIX$^{TM}$ command interpreter would allow a user to type "DNA-Reader < DNA-options > Display". This command would initiate execution of the program named "DNA-Reader" with its input from the file named "DNA-options" and its output directed to the device named "Display". Because the string "DNA-Reader" is a network name, the attempt to invoke it will automatically cause the local command interpreter to establish the connections noted above.

A supernetwork of the magnitude of NSF-net will remain with us for a long time. It is essential to have a clear vision of what the supernetwork will do in the long term, for otherwise it is easy to make early design decisions that will block important developments later. Obviously a supernetwork with the capabilities described here would be extremely powerful and would have a significant positive effect on scientific productivity. Such a system is within our technological grasp and NSF is reaching for it.

# Parallel Computation

*American Scientist 73*
No. 4 (Jul-Aug 1985)

*Computer systems that simultaneously apply the power of many processors to the solution of a single problem are coming of age. Why has it taken nearly 40 years for this to happen?*

In February, 1985, the Intel Corporation announced the iPSC$^{TM}$, a computer of a new structure, called a hypercube architecture, with the potential to surpass the fastest supercomputers at a fraction of the cost. The machine is based on a project called the Cosmic Cube headed by Chuck Seitz at Cal Tech (Seitz 1985). It consists of $2^n$ processor boards connected by a special high-speed network. (Intel currently offers machines for $n$ =5, 6, or 7.) The network connects processors as if they were on the corners of a cube in $n$ -space (hence the name, "hypercube"), which means that each processor is directly connected to $n$ others and that the longest path between any two processors spans $n$ links. A 128-processor machine is expected to sustain a processing rate of 10 MFLOPS (million floating-point operations per second) for the fastest algorithms, which is roughly one-fifteenth the sustained rate of the fastest real programs on the Cray X-MP-2X$^{TM}$ at one-twentieth the cost. (The maximal instantaneous speed of the Cray X-MP, in excess of 600 MFLOPS, is not sustained by real programs.)

The hypercube architecture has caught the fancy of many users of large-scale computing. Other manufacturers will soon offer competing products. Yet the concept of a computer containing many processors that simultaneously work on different parts of the same problem is as old as the era of electronic computing.

In the 1920s, Vanevar Bush of MIT demonstrated a general analog computer capable of solving arbitrary differential equations; his computer consisted of many components operating in parallel. The papers of von Neumann in the 1940s considered methods for solving differential equations on a discrete grid – all grid points were updated in parallel using the differential equation to determine how neighbors affect a particular grid point. Many models of possible computing substrates for intelligence were based on regular networks of automata. A substantial number of researchers during the 1960s considered a class of models called parallel program schemata; they sought to characterize the behavior of parallel systems and understand how to eliminate certain undesirable behaviors, such as indeterminate computations resulting from unpredictable speeds of components. The ILLIAC IV computer, constructed at the University of Illinois in the late 1960s, consisted of 64 processors operating in lock step; although its limited memory and expensive hardware prevented its proliferation, it inspired much interesting work in parallel algorithms. The emergence of very large scale integration (VLSI) technology in the 1970s stimulated interest in circuits composed of many parallel computers and in algorithms for using them.

There has thus been a continuing research effort to understand parallel computation. In spite of this, most parallel machines have been laboratory curiosities and most parallel algorithms paper studies. What has held this technology back from general commercialization for so long? Why is an idea that has lain relatively dormant for forty years now getting so much attention?

The answer lies in the strong conceptual simplicity of the sequential stored-program computer, the cost of processor technology, and a plausible argument that "bigger is better."

A sequential computer consists of a processor, a memory, and a communication subsystem. The processor fetches a sequence of instructions from memory. It decodes each instruction and carries out a specified operation on data in standard registers or in memory locations whose addresses are contained in the instruction. This approach to machine organization appeals to the strong intuitive idea of a step-by-step algorithm. It underlies the most common programming languages and much of the theory of computing. Its simplicity and universality make it a computer model of extremely wide appeal.

Not until the late 1970s was microelectronic technology mature enough that computer architects could seriously consider machines comprising many processors. Prototypes like the ILLIAC IV were based on technology that was unattractive for widespread use. Only recently has it become commercially feasible to build machines capable of comprising hundreds of processors.

Bigger-is-better arguments take a variety of forms. One is Grosch's Law, an empirical formulation dating back to the 1940s that says the cost of a computer in given technology is proportional to the square root of its speed. By extrapolation, a computer four times faster than this one would cost only twice as much. So why not seek the fastest possible computer?

Another form of the argument was pointed out to me by Len Kleinrock of UCLA. Suppose we have a supply of jobs requiring an average of $X$ computing operations each, and a processor capable of rate $R$ operations per second. The expected time to complete a job once started is $X/R$ seconds. Next, suppose we replace the single processor with $n$ identical, slower processors each with rate $R/n$. Let each job be partitioned into a chain of $n$ equal stages, the first stage being completed by the first processor, the second by the second processor, and so on. A total of $n$ jobs can be in various stages of execution in such a pipeline of processors. It is not hard to show that an $n$-stage pipeline has the same throughput as the single processor but $n$ times longer response time. While pipelining achieves parallelism, it is less responsive than a single processor of equal capacity.

A pipeline represents a "series decomposition" of processing power. We can also consider a "parallel decomposition" in which $n$ identical, processors of rate $R/n$ operate on whole jobs. In this case the parallelism maintains the throughput at the same value as for the single processor, but each job's response time is $n$ times larger. Again there is no apparent motivation for the slower processors.

There are two problems with the bigger-is-better arguments. The more fundamental is that there is a limit to the amount of computing power compressible into one box. The explanation, sometimes called the speed-of-light argument, goes like this. The speed of light is $3\times10^8$ m/sec in a vacuum and the signal transmission speed in silicon is at best $3\times10^7$ m/sec after gate switching delays are taken into account. A chip slightly over one inch in diameter, about 3 cm, can propagate a signal in about $10^{-9}$ second. Because a non-parallel chip can perform at most one floating-point operation during one signal-propagation, such a chip can support about 1 GFLOPS (giga floating-point operations per second).† For this reason microelectronics experts do not expect single processor machines in current technology to significantly exceed 1 GFLOPS in speed. Current supercomputers are within a factor of 10 of this limit.

This limit is becoming a serious problem because of the nonlinear relationship between a problem's size and the computing power required to solve it. For example, matrix multiplication takes about $n^3$ operations for $n\times n$ matrices. A problem twice as large thus requires a processor 8 times faster to complete in the same time. Equivalently, a processor twice as fast can multiply two matrices of size $n\times2^{1/3}$ in the same time, i.e., handle a problem only about 25% larger.

In other words, linear growth in our appetites to solve problems results in superlinear increases in the computing power required to solve them in the same amount of time. Sooner or later, we will require power beyond the capability of a single processor machine.

The second reason the bigger-is-better arguments break down is that they are critically dependent on the assumption about job partitioning. In the cases mentioned above, jobs are partitioned into sequences of components (pipelining) or are left intact (straight parallel execution). Consider, however, a more radical approach: Each job is broken into $n$ equal and independent components. Assume that the time to partition the input data among the $n$ components and aggregate the output results from the $n$ components is negligible compared to the time to complete a component. Now each component takes time

†The printed article erroneously reads "$10^9$ GFLOPS." It should read "$10^9$ FLOPS."

$(X/n)/(R/n) = X/R$ to complete on one of the slow processors. Because all components can be completed in the same interval, this system has the same response time and throughput as the single processor it replaces. There is a gain if each slow processor costs no more than $1/n$ of the cost of the single processor. (If Grosch's law applied, the total cost of the $n$ slow processors would be $\sqrt{n}$. larger than the cost of the single fast processor. But Grosch's law does not apply because different technologies are used: the fast processors tend to be handmade whereas cheap, slow processors tend to be mass-produced.) The conclusion is that the parallel decomposition can lead to a machine of the same throughput and response time at a fraction of the cost of a single processor – provided that the workload can be partitioned into independent, approximately equal pieces.

This proviso reveals that just beyond the speed barrier lies another: the software barrier. We're good at understanding sequential algorithms, but we have little experience with algorithms that direct and coordinate parellel processors. We don't know how to program the new machines. Most of the common programming languages are sequential – programs execute one statement at a time. The old computer languages, such as Fortran, Cobol, and Algol, and new languages, such as Pascal, share this property.

An easy way to think about parallel computation is a collection of separately running sequential programs, called processes, that exchange information among themselves via specific links. This gives rise to a model of parallel computation called communicating sequential processes (CSP) dating back to 1965. The programming language must be extended to include statements calling for the explicit creation of processes and for communication among them. The language Ada has such statements, as do Concurrent Pascal and a Pascal derivative called Occam.

The CSP model is more difficult to program than its close conceptual cousin, the sequential machine model. The reason is that the programmer must specify not one but many sequential processes and also the communications among them. The hypercube programmer, for example, is faced with the task of writing $2^n$ programs, one for each processor, and with understanding on the order of $(2^n)^2$ potential communication paths.

Understanding such a maze of possible interleaved executions and interactions is a formidable task for the unaided mind, not only because of the large number of possibilities but because of new types of errors that arise from interactions among processes. One type of potential problem is the nondeterminacy arising when two processes contain instructions to write a value in a shared variable: the value in the shared location will depend on which of the two processes wrote last. Another type of problem is the deadlock, where two or more processes get stuck in a cycle, each awaiting the receipt of information from another. The techniques for avoiding these problems are well known. Nondeterminacy can be avoided by forcing processes that share memory to proceed in some fixed order. Deadlock can be avoided by forcing processes to request information from one another in some fixed, global order. These techniques and the errors they prevent are unfamiliar to most programmers.

A major research goal in the years ahead is the development of new programming tools that aid the construction of systems of concurrent processes. These tools should operate at a higher level of abstraction than today's software tools. With them, the effort to specify a correct concurrent computation should be no greater than the effort today to specify a single sequential process.

An example of such a system is the Programming Environment known as "Poker" being developed in Larry Snyder's CHiP project at the University of Washington (Snyder 1984). Poker presents a graphics screen containing a block diagram of an array of processors. The programmer specifies desired interconnections by drawing links between some of the processors. The programmer specifies programs by entering them in a language (e.g., Pascal) and pointing to the boxes of the processors that are to run them. Poker handles the details of loading the programs into the processors, inserting the correct communication statements into the compiled programs, and automatically establishing all the required links. Poker can also be extended to automatically enforce rules preventing nondeterminacy or deadlock. The effort to construct a system of parallel programs using Poker is about the same as that required to construct one Pascal program using conventional programming aids.

We are near the limits of single-processor technology. The only available approach to computing power much beyond 1 GFLOPS is machines consisting of many parallel processors coupled with new programming methods that allow partitioning jobs into many

## *References*

Seitz, C. 1985. "The Cosmic Cube." *Communications* of the ACM, January.

Snyder, L. 1984. "Parallel Programming and The Poker Programming Environment." IEEE *Computer*, July.

# The Evolution of Parallel Processing

*American Scientist 73*
No. 5 (Sep-Oct 1985)

*The software barriers to parallel computation are formidable. Progress toward computer systems that exploit parallelism is likely to be slow. It will be governed by the rate at which we can find tools to aid the most abstract aspects of problem-solving.*

The software barrier – the set of limitations in software technology that impedes our effective use of parallel hardware technology – is not a brick wall that can be broken down with a heavy ram. It is a deep, thick jungle through which slow progress will be achieved by constant chopping and hacking. I would like to describe the four stages of our probable journey through this torturous territory. Stage 1 is nearly finished and Stage 2 is under way. Tentative explorations are beginning on Stage 3. Stages 4 is more distant.

IN STAGE ONE, parallelism is introduced into the hardware of a single computer, which consists of one or more processors, a main storage system, a secondary storage system, and various peripheral devices. Old examples of this include multiple function units in the processor, multiple-bank memory, pipeline execution of instructions, and vector pipelines that apply one operation to a stream of data. These old ideas are undergoing a new round of improvements in a class of machines called reduced instruction set computers (RISCs). RISCs are fast partly because their instruction sets are small and partly because their compilers carefully analyze programs to create code patterns that keep the instruction pipeline busy most of the time.

The computer hardware is controlled by a program called the operating system. Most operating systems contain a ready queue that lists the names of all processes (programs in execution) awaiting their turns to run on a processor. With a few changes, an operating system can be extended so that multiple processors serve the ready queue rather than just one. This principle is exploited by a genre of machines called symmetric multiprocessors. As early as the late 1960s, a few manufacturers offered such machines with up to four processors. Today there are commercial multiprocessors with as many as two dozen processors; the capacities of these machines can be increased simply by plugging in more processor boards. A more radical multiprocessor is the NYU Ultracomputer, an experimental project undertaken jointly by New York University and IBM; new processors and memory can be added indefinitely without saturating the processor-memory interface.

Stage 1 is relatively easy on users because it requires almost no change to the visible software technology. For example, the Cray-1's vector hardware can be exploited by reorganizing algorithms with minimal changes in the Fortran language once the Fortran compiler has been updated to handle vectors of data. Similarly, key subroutines, such as Fast Fourier Transform, can be recoded for array or vector machines without changing the calling protocols. A multiprocessor can be exploited without changes in the UNIX$^{TM}$ operating system's command language once the UNIX kernel has been updated to allow multiple processors to execute in it simultaneously. Thus, the software changes needed for Stage 1 parallelism can be confined to the compilers, the libraries, and the operating system and are largely invisible to users of high level languages.

Stage 1 is now well under way and will be mature within a few years. Most program codes include so many assumptions about sequential execution that they, rather than the hardware, limit our ability to use up the power of parallel machines. Our hunger for more computing power will force us to Stage 2.

IN STAGE TWO, parallel execution of cooperating programs on different machines becomes explicit. Programs exchange data over high speed communication links rather than by passing addresses of shared data segments. The hypercube architecture, such as in the Intel iPSC$^{TM}$, is of this type.

A few changes in programming language are needed at Stage 2. Because there is no common store, processes operating on different machines cannot exchange data by accessing shared variables. Instead they must invoke network commands. An especially simple notation for this was proposed by C. A. R. Hoare of the University of Oxford [Hoare, 1978]. A new type of variable, called a port, is added to the programming language. At a place where a value is needed from another process, the program contains the port name followed by a question mark. Where a value is to be sent, the program contains the port name preceded by an exclamation point. The sender and receiver must rendezvous at a port before the data is actually transferred between them. For example, a process that reads values $X$ and $Y$ from input ports and transmits the sum to an output port $Z$ would contain the statement

$$!Z = X?+Y?$$

In evaluating this expression, the process waits until both ports $X$ and $Y$ contain values, then sums them, then sends the result on port $Z$. These notations are contained in a derivative of Pascal called Occam.

More extensive changes are taking place in operating systems. Instead of managing one computer, operating systems are being extended to manage networks of computers. The interface will be kept simple so that the same operations will deal with resources whether they are local or remote. These "distributed operating systems" will eventually allow the construction of computations that span many machines, of different types (e.g., workstations and supercomputers).

To help programmers keep track of a potentially large number of interacting programs and machines, and to properly load programs into their machines at run time, we will need new software development tools. An example is the Poker programming system described in my previous column.

Although not yet used widely, these changes in software technology are well under way. Progress in Stage 2 will be limited more by algorithm technology than by programming technology. How can kernel codes for fluid dynamics, chemical properties, finite structural analysis, seismic modeling, or petroleum exploration be decomposed into parts that can be run on separate machines? Can the numerical properties and stability of algorithms that solve linear equations or partial different equations be preserved under such partitions? How extensively will our basic mathematical and statistical software libraries need to be reworked for partitioned machines? Ahead may lie a substantial effort to exploit new knowledge about basic algorithms in real codes.

I expect a lot of algorithms research during Stage 2. A great deal of detailed information about algorithms for partitioned machines will be discovered. But the complexity of what is learned will create a strong pressure to hide the details behind simple interfaces and to find compilers and operating systems that can perform algorithm-partitioning automatically. This will force us toward Stage 3.

IN STAGE THREE, new languages will make parallelism implicit and their compilers will take over the burden of partitioning programs. Conventional programming languages are imperative in the sense that their statements are all treated as commands directing a processor. In contrast, the new languages are functional, which means their statements are treated as expressions denoting compositions of functions. Examples are:

1. LISP, which specializes in the manipulation of strings of symbols that denote expressions and values;
2. VAL and LUCID, which specialize in dataflow computations, where operations fire as soon as all their operands are available;
3. REDUCE and MACSYMA, which specialize in symbolic algebraic expressions – differentiation, integration, and reduction to minimal terms;

4. SQL, which specializes in queries of relational databases;
5. APL, which specializes in applying functions to vectors of data; and
6. PROLOG, which specializes in evaluating expressions in a deductive logical system.
7. This list of examples is not exhaustive. the common feature of these languages is that they describe the results of a computation but not the method of obtaining the results. this leaves considerable flexibility for compilers and operating systems to distribute pieces of a computation among many processing elements.

to date there is very little experience with these languages outside of computer science. there is good reason to believe that they are incomplete with respect to solving real problems in other disciplines. many problems decompose naturally into pieces called chunks that can be solved separately and must therefore be accounted for explicitly during the formulation of an algorithm. different chunks may require solution on different machines and in different languages. an example of this arises in computational fluid dynamics, where a test region may be divided into zones, each zone being treated with a different algorithm. Not only is each zone a chunk in its own right, but each zone may be composed of dissimilar subchunks – for example a chunk to derive symbolic equations from the zone's description, a chunk containing a symbolic manipulator to reduce the equations to minimal terms and generate a corresponding Fortran program, and a chunk to execute the Fortran code. The inability of Stage 3 languages to deal with heterogeneous chunks, or to define chunk boundaries, will motivate progress toward Stage 4.

IN STAGE FOUR, there will be very high level user interfaces capable of interacting with scientists at the same level of abstraction as scientists do with each other. These interfaces will help formulate precise descriptions of problems in a given domain, using natural language, pictures, speech, and formal notation. The interface systems will convert these descriptions into natural chunks, construct functional descriptions of the chunks, convert each functional description to a specific program, convert each program to executable code, request code execution, and finally collect the results for display at the abstraction level of the domain. Stage 4 systems will use today's expert-system technology as a building tool. This technology facilitates storage and use of rules stating the desired response when given stimuli are presented.

There is A striking resemblance between the probable stages of evolution of computing technology for science and the hierarchy of abstractions in the process of formulating computational solutions to scientific problems. From the most abstract to the most specific levels, the hierarchy is:

4. PRECISE DESCRIPTION. Construct a precise description of the model for the problem and the constraints to be observed during solution. This description may use natural language, mathematical notation, and special terminology and notations of the discipline.
3. ABSTRACT ALGORITHMS. Specify the general strategy to be used to solve the problem according to the given description. How will the solution partition into chunks? What strategies of iteration, data exchange, and convergence will be used? (At this level we do not worry about details like data representation or machine capacity.)
2. CONCRETE ALGORITHMS. Construct or hook together program modules for the various pieces of the solution. Each module may be represented in a different language.
1. MACHINE CODES. Construct codes in the instruction sets of machines and distributed operating systems to carry out the detailed computations.

The problem-solving process is a series of transformations from the precise description at the abstraction level of the discipline down to the precise description of an algorithm at the level of abstraction of the hardware.

The four stages of evolution toward parallel computation correspond one-for-one with the four principal abstraction levels in the problem-solving process. Each stage of evolution is a step in the development of tools for programming at a particular level of this hierarchy. Computer science currently offers few tools to support transformations at the higher levels of the hierarchy: this is why few Stage 3 or Stage 4 systems exist. Although parallel computation is less visible at the higher higher levels, it is no less important: it gives the means to implement the performance needed to operate systems at Stages 3 and 4.

## *Reference*

Hoare, C. A. R. 1978. "Communicating sequential processes." *Communications of ACM*, August.

# The Arbitration Problem

*American Scientist 73*
No. 6 (Nov-Dec 1985)

***Deep in a computer's hardware are circuits called arbiters whose function is to select exactly one out of a set of binary signals. If one of the signals can change from "0" to "1" while the selection is being made, the subsequent behavior of the computer may be unpredictable. It appears fundamentally impossible to construct an arbiter that can reliably make its selection within a bounded time interval.***

The 14th century philosopher Jean Buridan described the paradox of the hungry dog who, being placed midway between two equal portions of food, starved (Rescher 1967). Everyone has experienced the sensation of frozen immobility when faced with a choice between two equally appealing alternatives. Who would think that this problem is so fundamental that it limits the ability of computer circuits and software to reliably resolve contention for shared resources?

Computers contain circuits called arbiters for selecting one of many potential requests for serially reusable resources, which can be used by only one processor at a time. Common hardware devices such as memory banks, arithmetic function units, and communication channels are resources of this type. Some software, such as files of data and subroutines with private internal variables, is also of this type. What happens if an arbiter must select among near-simultaneous signals?

If, like the ill-fated dog, the arbiter fails to decide on a selection within the time limit assumed by other circuits, the results can be devastating. One form of arbitration failure is that no processor gains access and the computer stops; another is that several processors gain access at once, producing an inconsistent state of the shared resource and leading eventually to complete system failure (a "crash"). An apparently small probability of arbiter failure can still be significant because, at computer speeds, large numbers of arbitration events can occur in a short time.

Chuck Seitz of Caltech gives an excellent description of arbitration failure for very large scale integrated circuits (Seitz 1980). The following discussion barely scratches the surface of this nugget. An $N$-way arbiter has $N$ inputs (one per processor). It must have exactly $N+1$ externally observable, stable output states. State $i$ (for $i = 1,...,N$) signifies that processor $i$ has been selected and is using the shared resource; state 0 signifies that no processor is selected and the resource is free. If processor $i$ signals when the arbiter is in state 0, the arbiter will move to state $i$, possibly through a chain of transient internal states. If processor $i$ signals when the arbiter is not in state 0, the signal is saved and the processor waits. When the resource is released by processor $i$, the arbiter returns to state 0 if no signals are waiting and moves to another state $j$ otherwise. Thus it is easy to follow the operation of the arbiter when processors supply inputs at distinguishably different times.

The trouble begins if two processors signal their requests almost simultaneously – i.e., within the selection interval. (The selection interval, $\Delta$, is on the order of $10^{-10}$ to $10^{-11}$ seconds in today's technology.) In this case, the arbiter is simultaneously asked to follow the trajectories $0 \rightarrow i$ and $0 \rightarrow j$ for distinct $i$ and $j$. It can now wind up in an internal state poised exquisitely and equally between the two targets, $i$ and $j$, where it will remain until dislodged by noise or another input. This unstable equilibrium is called a *metastable state*. As long as the arbiter is in a metastable state, its output lines, which contain codes derived from the internal conditions associated with stable states, are meaningless. The output voltages might assume intermediate values that cannot be reliably interpreted as binary "0" or "1"; they might appear to indicate two or more selections; they might oscillate.

Metastable states are not the same as transient states that may be visited during normal transitions between stable states. A metastable state is a point of equilibrium that can persist indefinitely; which stable state is entered next is unpredictable. A transient state, on the other hand, is not a point of equilibrium; it has a short, bounded holding time.

The time it takes to reenter a stable state depends on how close to a metastable state the arbiter was driven by its inputs and on the arbiter's switching speed. (Noise affects the position of the metastable equilibrium point but not the probability of a particular duration.) The situation is analogous to that of a ball set motionless atop a rock. The time until the ball falls from the rock and comes to rest on the ground depends on how finely balanced it was on the rock's pinnacle and on the curvature of the rock. Just as it is impossible to predict exactly how long the ball will stay in place on the rock, it is impossible to predict exactly how long the metastable state of the arbiter will persist.

Analyses of the differential equations of arbiter circuits show a common pattern. The probability that the metastable state lasts at least $t$ seconds is $e^{-\alpha t}$ for a parameter $\alpha$ that depends on the circuit technology. Typically $1/\alpha$ is on the order of $10^{-9}$ second, the circuit's switching time. If $f$ arbitrations per second are needed and the minimum distinguishable separation between signals is $\Delta$ seconds, the probability a given arbitration event pushes the arbiter into a metastable state that persists $n = t/\alpha$ circuit switching times is $f \Delta e^{-n}$. Seitz gives an example of a disk generating $10^6$ interrupts per second in a circuit with $\Delta = 10^{-11}$ second and $n = 16$: arbitration failure will occur about once in every $10^{12}$ interrupts, which comes out to about once every 10 days.

In the 1970s, Thomas Chaney and Charles Molnar of Washington University in St. Louis conducted studies of arbiter circuits under repeated applications of near-simultaneous inputs. Their group produced photographs of oscilloscope traces showing circuits stuck in their metastable states for amazingly long periods, some up to about 30 times $1/\alpha$. No circuit is known that avoids metastable behavior when subjected to such tests.

Is there any way to avoid arbitration failure? It is tempting to say: Yes, run everything off a common clock; stagger the times at which signals are sampled and the times at which signals are used. Then the using circuits see only unchanging signals emitted by the sampling circuits. The problem with this hypothesis is that it is impossible to control when some signals will arrive. For example, the interrupt signal announcing the end of a file transfer cannot be precisely controlled because the disk rotation time cannot be precisely controlled. In general, whenever the timing of at least one of two confluent signals cannot be precisely controlled, arbitration failure can occur at their junction.

So there is no solution to arbitration based on a continuously running clock. What happens if the clock can be stopped? David Wheeler of the Computing Laboratory at the University of Cambridge (England) designed an arbiter for the Cambridge CAP computer. Wheeler's arbiter contains an additional threshold circuit that detects when a stable state has been entered and then produces a special output. By design, the companion circuits shut off the clock just after initiating an arbitration event; the arbitration circuit restarts the clock as soon as the special output signals that the arbiter has settled. Now arbitration failure is impossible. But there is a price: the special output cannot be guaranteed to appear within any preset, bounded interval. (In fact, the probability the special output is still off after $n$ circuit-switching times is on the order of $e^{-n}$.) On average, Wheeler's circuit takes time $1/\alpha$ to reach its decision – a small loss in average speed. Nonetheless a few decisions may take a long time that may exceed the tolerances of some systems.

Seitz also describes a class of circuits that use no clock at all, called self-timed circuits. Self-timed circuit elements interact by exchanging request and acknowledge signals; no element can generate a next request until after it has received an acknlowledgement for its previous request. In this context, an element called an interlock, which resembles Wheeler's circuit, is used for arbitration. An interlock cannot cause arbitration failures, because neighboring circuits are forced to wait until the interlock has reached a decision. Because they tend to consume more energy in interelement signalling than clocked circuits, self-timed circuits have been less attractive for high performance computers. However, their greater reliability makes them increasingly attractive for massively parallel computers.

Arbiters and interlocks are also called mutual exclusion circuits. They are examples of circuits used to synchronize unclocked inputs with the internal clock. Synchronizers must contain arbiters to handle the case of inputs changing at the same time as the clock. For this reason, arbitration failure is also called synchronization failure.

It might seem that arbitration failure is a problem that needs to be understood and solved by hardware designers but not by software designers. Nothing could be farther from the truth. Programmers of parallel computations are constantly faced with the challenge of finding ways to avoid arbitration failures when parallel tasks compete for shared data.

In 1965 Edsgar Dijkstra of the Technische Hogeschool Eindhoven published a paper that foresaw the software problems caused by attempts to control parallel processors (Dijkstra, 1965). Entire books have since been devoted to the subject and some modern programming languages include syntax for dealing with the problem. Dijkstra's formulation of the problem was this: Consider a computer consisting of $N$ processors having access to a single shared memory. Each processor runs its own program. Each program contains a critical section of code whose instructions examine and modify data that is accessible to all the processors. Two processors must not be allowed simultaneous access to the shared data lest they interfere with each other and produce inconsistent results. Is it possible to program common entry and exit protocols to critical sections in such a way that this "mutual exclusion" requirement is met no matter what the relative speeds of execution of the processors? The entry and exit protocols should be the same and contain no built-in knowledge of which processors might use them.

A simple example will illustrate the interference problem Dijkstra ought a way to avoid. In commercial banks, automatic teller machine programs contains a subroutine to transfer funds between accounts; a transfer of $X$ dollars from account $A$ to account $B$ might be expressed by the program statements "$A = A - X$; $B = B + X$". Execution of these statements is supposed to change the state of the accounts from ($A$, $B$) to ($A - X$, $B + X$), leaving the sum $A + B$ unaltered. Now: suppose I initiate a transfer of \$1000 from $A$ to $B$ while (from a different teller machine) my wife initiates a transfer of \$400 from $B$ to $A$. No matter which order we perform our transactions, the resulting state of our accounts should be ($A$ −600, $B$ +600). But suppose our two processors can execute the machine code for these statements simultaneously. The following events can occur. Just after my processor writes $A$ −1000 and before it writes $B$ +1000, my wife's processor can read the account values, seeing state ($A$ −1000, $B$). When her processor later writes its results, it overwrites the value $B$ +1000 subsequently written by my processor , leaving the state of accounts as ($A$ −600, $B$ −400) – a loss of \$1000. Although the critical sections appear as units in the programming language, they are implemented as machine code sequences. Without the guarantee of mutual exclusion at the programming language level, these code sequences can be interleaved during execution, leading to results that were not anticipated by the programmer.

In solving the problem of making critical sections indivisible, Dijkstra took advantage of the fact that memory hardware contains arbiters that would allow only one processor at a time to gain access to memory. From the programmer's viewpoint, this means that reading or writing scalar variables is an indivisible operation. Dijkstra showed how to implement two programs, ENTER and EXIT, that used shared scalar variables as signals among the processors. He proved rigorously that these programs guaranteed mutual exclusion of any critical section they enclosed. He proved, moreover, that this solution was not vacuous because within a bounded time after one processor released its critical section, another would complete the ENTER procedure.

Dijkstra's programs were complicated and difficult to understand. Yet the gravity of the arbitration problem these programs solved was not hard to understand. So computer architects devised a more direct solution. The hardware solution is to create a special instruction that indivisibly sets a lock. This one instruction accomplishes the same work as did a loop examining the signal variables in Dijkstra's ENTER procedure.

The instruction "LOCK $X$" treats memory address $X$ as a binary variable. If it finds $X=0$ it sets $X=1$ in the same memory cycle and proceeds. If it finds $X=1$ it retries the instruction in a later memory cycle. Dijktra's ENTER operation can be programmed simply as "LOCK $X$" for a signal variable associated with the critical operations. His EXIT instruction is an ordinary "CLEAR $X$".

Unfortunately, this solution has a serious limitation that prevents its extension to large numbers of processors. If many processors seek access to their critical sections at the same time, all but one get stuck in a loop retrying the LOCK instruction until one of them reads the $X=0$ left by the processor exiting the critical section. (The arbiter circuit guarantees that only one gains access to the memory after $X$ becomes 0.) The problem is that each cycle of this loop requires an access to memory (to observe that $X$ is still 1). During this cycle any processor seeking access to other variables stored in the same memory is blocked. This means that the lock-testing loop will interfere with the progress of any processor doing legitimate work. As the number of processors in the system increases the overhead of lock testing increases, to the point finally of preventing any new processor from doing any useful work.

Two solutions to the problem of excessive lock testing are used in modern multiprocessor computer systems. The first, proposed by Dijkstra in 1965, is to replace the lock with a queue called a semaphore. Any processor attempting the ENTER protocol when the semaphore is in use is suspended and its name placed in the queue. The EXIT protocol releases one of the enqueued processors, if any (Denning *et al.* 1981). In the second solution, each processor has a local memory called a cache that contains copies of values in the main memory. Thus the processor executing a LOCK $X$ instruction gets a copy of the lock variable $X$ in its cache and can loop there without interfering with any other processor. The CLEAR instruction broadcasts the message "$X = 0$", which is captured by one and only one of the caches if one is busy waiting for the lock or by the memory if no cache is waiting. Both these solutions reduce lock contention and allow a moderate number of processors to share memory.

It is actually more efficient to lock individual resources rather than critical sections. This allows two processors to run concurrently in their critical sections if they are using different resources. In the banking example, we would use separate locks for the accounts $A$ and $B$ rather than one lock on the transfer subroutine.

The software solutions to the arbitration problem rest ultimately on the hardware arbiter that decides which processor gets the memory during the next memory cycle. The reliability of software arbitration is therefore limited by the reliability of the arbiter.

## *References:*

Rescher, N. 1967. "Buridan, Jean." In *Encyclopedia of Philosophy*. MacMillan.

Denning, P. J., Dennis, T. D., and Brumfield, J. A. 1981. "Low Contention Semaphores and Ready Lists." *Communications of ACM*, October.

Dijkstra, E. W. 1965. "Solution of a problem in concurrent process control." *Communications* of the ACM, September, p. 569.

Seitz, C. L. 1980. Chapter 7, "System Timing." In *Introduction to VLSI Systems* by C. Mead and L. Conway, Addison-Wesley.